User's Manual as a Requirements Specification

Daniel M. Berry, 1991, 1998, 2002, and 2003 with help from K. Daudjee, J. Dong, I. Fainchtein, M.A. Nelson, T. Nelson, & L. Ou

© 2003 Daniel M. Berry

A Note About This Talk

This talk was first written in 1991, prior to the community's identification of the concepts of *scenarios* and *use cases*, as being helpful in requirements engineering (RE).

It is interesting to see a lot of examples of these concepts in these slides, but not so-named.

I have added text in this font in 1998, to rephrase in the new vocabulary.

Introduction -1

I have been asked what I believe a good requirements specification (RS) is.

Here is my opinion.

For pedagogic purposes, it *is* intended to be slightly controversial.

The "you" in these slides is the requirement engineer.

Introduction -2

I believe that the most useful document to write during requirements engineering is the user's manual (UM).

When done right and at the right time, it can serve as a useful elicitation, analysis, and validation tool, and can even serve as a RS.

Introduction -3

Please note that I am not discounting the other requirements documents, including the SRS.

They may be required by the customer.

They may give useful information that is not contained in the UM.

Introduction -4

However, I have found the production of the UM a good focal point in the requirements engineering process and a good way to get on to paper at least the kernel of all the information that goes in the other documents.

Role of Writing a RS -1

There are a number of reasons that writing a RS for a computer-based system (CBS) before implementing it is a good idea.

- **The process of writing the RS of the CBS is a good way to learn the CBS's requirements.**
- **The process of writing the RS of the CBS helps to reconcile differences among the CBS's stakeholders.**

Role of Writing a RS -2

- **The RS allows the customer of the CBS to validate that the projected CBS will be what he or she wants before resources are spent implementing a possibly incorrect CBS.**
- **The RS makes it clear what must be implemented to obtain the required CBS.**
- **The RS allows deriving both covering test cases and expected results that allow verification that the implementation of the CBS does what it is supposed to do.**

Problems Writing a Good RS -1

Despite clear benefits of writing a RS for a CBS before implementing it, many projects are unable to produce the RS, for a variety of reasons, some technical and some social.

- **It is difficult to write a good RS, one that specifies exactly what the CBS is supposed to do without limiting unnecessarily how to implement it.**

Problems Writing a Good RS -2

- **Participants in most projects these days believe that they do not have the time to do so, that it is necessary to proceed immediately, if not before, to coding, in order to meet the code's delivery deadline or to be the first in the market with the code's functionality (Never mind how they know what to implement anyway if there is no RS).**

Problems Writing a Good RS -3

- **Participants in most projects these days perceive that time spent on writing a RS is wasted since the requirements will change anyway and the RS will probably never be read, even by the implementers.**

Writing UM May Be Solution

This talk offers writing a UM for a CBS before implementing it as a way achieving the writing of a RS of the CBS before implementing it.

The method both

- **produces a document that delivers the five benefits of writing a RS before implementation and**
- **helps ameliorate or mitigate the three problems that discourage the production of a RS before implementation.**

Information in a UM -1

An RS for a CBS should

- **describe the CBS's function and not the CBS's implementation,**
- **describe the CBS from the user's point of view and not the implementer's,**

Information in a UM -2

A good UM for a CBS should

- **describe the CBS's function and not the CBS's implementation,**
- **describe the CBS from the user's point of view and not the implementer's,**

Hmmmm????

Fred Brooks's Observation

In 1975, in *MM-M*, Fred Brooks equated the UM with the written RS:

The manual must not only describe everything the user does see, including all interfaces; it must also refrain from describing what the user does not see. That is the implementer's business, and there his design freedom must be unconstrained. The architect must always be prepared to show* an *implementation for any feature he describes, but he must not attempt to dictate* the *implementation.

Tom DeMarco

Also, Tom DeMarco suggests in several places using UMs as RSs, most notably in *The Deadline*.

Steve McConnell

In *Software Project Survival Guide*, Steve McConnell says, metime prior to placing the prototype under change control, work can begin on a detailed user documentation (called the User Manual/Requirements Specification). This is the documentation that will eventually be delivered to the software's end users. Typically, this documentation is developed at the end of the project, but in this book's approach, it is developed near the beginning."

Lisa & Macintosh

It is said that the UMs for the Lisa and Macintosh computers were written completely before implementation of their software began.

The UMs were given to systems programmers as the RS of the user interfaces (UIs) and of the underlying systems.

UMs and 5 Roles of a RS -1

I claim that

- **The process of writing a UM for a CBS is a good way to learn the CBS's requirements.**
- **The process of writing a UM for a CBS helps to reconcile differences among the CBS's stakeholders.**
- **A UM allows the customer of the CBS to validate that the projected CBS will be what he or she wants before resources are spent implementing a possibly incorrect CBS.**

UMs and 5 Roles of a RS -2

- **A UM makes it clear what must be implemented to obtain the required CBS.**
- **A UM allows deriving both covering test cases and expected results that allow verification that the implementation of the CBS does what it is supposed to do.**

Motivating Writing of a RS
Difficult to Write a Good RS -1

Writing a good RS for a CBS is difficult because the advice to describe only what the CBS does, without constraining how the CBS does it, is easier said than done.

Clients and users tend to describe solutions to possibly non-existent problems rather than just problems that need to be solved.

Difficult to Write a Good RS -1

Writing a good UM for a CBS forces focusing on user's view of the CBS.

With typical user in mind as the future audience of the UM, it's easier to focus on the user's view, the what of the CBS, and to avoid mentioning implementation details.

Not Enough Time to Write a RS
Writing a RS is a Waste of Time

If a project produces a UM or help system or it produces test cases, it writes a RS.

Any project for commercial or contracted software does so.

Thus there *is* time to write a RS; it's already being done.

Not Enough Time/Time Waster

Ah, but the UM, help system, or test cases are written later.

Ah, but writing them earlier *saves* time and money for each requirement error found earlier when it costs an order of magnitude less to fix it.

Writing UM Preferable to Writing RS

UM or help system needs to be written eventually for user's benefit, but RS is not likely to be looked at beyond beginning of coding.

Good UM exposes full set of use cases, from which test cases can be written, but most SRSs tend to focus on functional and nonfunctional requirements (NFRs) and skip use cases.

Requirements for RS -1

It is most important that a requirements document

- **be readable, and**
- **accurately and completely describe the software system to be built, a system that meets the client's desires and needs.**

All else is frosting on the cake.

Requirements for RS -2

It must be readable because if not,

- **no one will be able to judge whether the document meets the second document requirement,**
- **no one will be able to write the software to meet the system requirements,**
- **no one will be able to judge whether the software meets the system requirements.**

Good UMs -1

My favorite way to write a RS for a system that will have users is to write a UM or a collection of UMs, one for each kind of user.

Good UMs -2

Writing UM requires a clear conception of what the system is supposed to do, clear enough that the manual author can visualize user scenarios and describe both

- **what the user should say to the system and**
- **what the system will respond to the user.**

Note: Scenarios!

Good UMs -3

So what does a good UM look like?

Well, it should be clear, not longer than necessary, and fun to read!

Actually, almost everyone knows a bad user manual when he or she tries to read one and cannot!

There is something of an art to writing a good UM.

Good UMs -4

I personally have found the following manuals good:

- **the *PARADOX UM* from Borland**
- **the *TEXbook* by Knuth**
- **the *C Programming Language* by Kernighan and Ritchie**
- **the *PIC UM* by Kernighan**
- **the *EQN UM* by Kernighan and Cherry**

Bad UMs -5

I personally have found the following manuals bad

- **the *C++ Programming Language* by Stroustrup**
- **the *NROFF/TROFF UM* by Ossana**
- **the *SCRIBE UM* by Unilogic**
- **the *TBL UM* by Lesk**

Good UMs -6

A good UM seems to have the following elements:

1. **descriptions of underlying and fundamental concepts of the software,**

i.e., a Lexicon!

Good UMs -7

2. **a graduated set of examples each showing**
   - **a problem situation the user faces**
   - **some possible user responses to the problem in the form of commands to the software**
   - **the software's response to these commands**

   i.e., Use cases!

3. **a systematic summary of all the commands**

Good UMs -8

Having only the third loses many readers who do not understand the concepts and turns off many readers who just plain get bored reading page after page after page of boring command syntax and semantics.

Leaving out the first makes it very hard for the author to assume and use a consistent vocabulary in writing the rest of the manual.

Good UMs -9

Leaving out the second leaves the reader without any sense of what is important and how to use the system to solve his or her problems.

A well-written second part makes reading the manual, even the third part, fun.

Good UMs -10

The third part must be consulted in order to fully explain why the input of an example solved the problem the example claims it does.

It is in writing the first and second parts that diagrams are most useful, although I have seen good third parts that use a collection of related diagrams, one per command, to explain the system response to every command.

Good UMs -11

A good way to organize the first part is around the abstractions that you have found in the problem domain.

i.e., a Domain Model!

Each abstraction that survives the analysis should be explained in terms of

- **what the objects are**
- **what they do**
- **what is done to them**

Good UMs -12

Writing a good UM takes skill, and there is no substitute for that skill.

I hope that at least one person in each group has that skill.

In an industrial situation, the client and the software house must hire good writers to write skillful conception and requirements documents.

English Majors as Documenters

Bob Glass reports how successfully non-software-knowledgeable English majors were able to write high-quality descriptions of the grubby details of programs in documentation about these programs for maintainers.

Another Opinion

According to Richard Fairley in his 1985 *Software Engineering Concepts*, a preliminary UM should be produced at requirements definition time.

He proposes the following outline for the (preliminary) manual.

Preliminary UM -1

1. **Introduction**
   - **Product overview and rationale**
   - **Terminology and basic features**
   - **Summary of display and report formats**
   - **Outline of the manual**

Preliminary UM -2

2. **Getting started**
   - **Sign-on**
   - **Help mode**
   - **Sample run**
3. **Modes of operation: Commands/Dialogues/Reports**
4. **Advanced features**
5. **Command syntax and system options**

Where are the Scenarios?

Chapter 3, about Modes of Operation, is precisely a list of scenarios, dressed up as a list of

- problems the user might be faced with, and
- how to solve them with the software being described.

CBSs Admitting UMs as RSs -1

The approach of offering a UM as the principal or only RS of a CBS works only for those CBSs for which a UM describes all but trivially explained requirements.

CBSs Admitting UMs as RSs -2

Thus,

- **The CBS must have at least one kind of user.**
- **The CBS must provide all but trivially explained functionality through at least one UI, and all of this functionality must be clear from the behavior seen by a user.**
- **Each of the CBS's NFRs must be well understood and easily described in prose.**

CBSs Admitting UMs as RSs -3

If a CBS has several kinds of users, one UMs manual can be written for each kind.

However, then achieving and maintaining consistency of all UMs becomes a problem.

CBSs Not Admitting UM RSs -1

- **Autonomous systems with no real human users: However, a description of the real world's or other CBS's behavior might suffice.**
- **A CBS for which one or more algorithms it computes is the major issue of a RS (and the UI is not an issue): e.g., a weather predictor**

CBSs Not Admitting UM RSs -2

- **A CBS with nontrivial NFRs that are not specifically visible to the user, e.g., security, reliability, and robustness (SR&R), for which the user does nothing to cause the NFR to kick in. The way SR&R is achieved is a major issue for the RS.**

For none of these CBSs would a UMs manual serve as a good RS.

Validating Case Studies

1. **Development of FLO (Berry) Academic Product**
2. **Development of WD-PIC (Berry; Berry, Daudjee, Dong, & NelsonS; Ou) Academic Product**
3. **Development of ExpressPath (Fainchtein) Industrial Product**

FLO Experience

Let me describe an experience assisting in the writing of a UM as a primary requirements document.

This experience had an interesting twist to it that cannot happen for all software, and probably will not happen for yours; however it is interesting from the total software engineering perspective.

FLO—A Language For Typesetting Flowcharts

Tony Wolfman
Daniel M. Berry, 1989

Outline

- problem
- previous solutions
- goals
- concept
  includable flowchart
- design approach
  design | implementation | testing
  of
  language | program | manual

Problem -1

It is desired to be able to include flowcharts as figures inside typeset documents,

e.g., as for publishable papers in computer science or in software descriptions.

Input: algorithm

Program draws flowchart

Problem -2

We assume the following basic typesetting software:

- device independent TROFF (DITROFF)
- but if we do it right, it will be usable elsewhere, e.g, in TEX

Previous Solutions -1

1. Computer produced flowcharts (1960s):

- Aim—provide better maintainer's documentation and debugging facilities
- Input—a source program in some programming language
- Output—large, very complicated flowchart encompassing many pages, usually produced on line-printer

Note, these programs had to be general enough to handle *any* program!

Previous Solutions -2

2. Programs for typesetting graphs:

- **Aim—include general graphs into documents**
- **Input—connectivity of the graph**
- **Output—some picture description language for typesetting, e.g., POSTSCRIPT or PIC**

Previous Solutions -3

3. Interactive WYSIWYG drawing programs:

- **Aim—draw picture for inclusion into typeset documents**
- **Input—interactive, mouse movements, etc.**
- **Output—some picture description language for typesetting, e.g., POSTSCRIPT or PIC**

Previous Solutions -4

The problems with these previous solutions are:

1: The output is not satisfactory for inclusion in typeset documents (boxes built with "l" and "_" and using numbered nodes instead of some nicely drawn arrows, especially when the arrows would cross page boundaries).

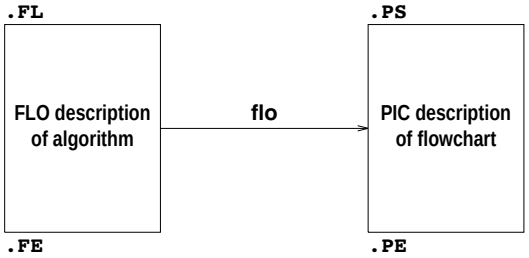
Previous Solutions -5

2: The user must input the topology of flowchart and not the algorithm of flowchart.

3: Same as 2!

Goals -1

1. **be able to include flowcharts into documents typeset with DITROFF via PIC**

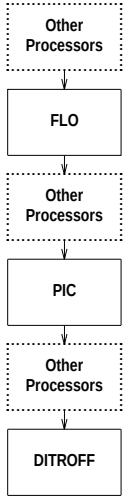


Goals -2

The one page limit of PIC pictures leads to concept of includable flowchart, a flowchart that can be shown entirely on one page.

If the user needs a larger flowchart, he or she must manually break it into page-sized subflowcharts and specify the interpage connections explicitly.

The next slide shows the DITROFF pipe:



Goals -3

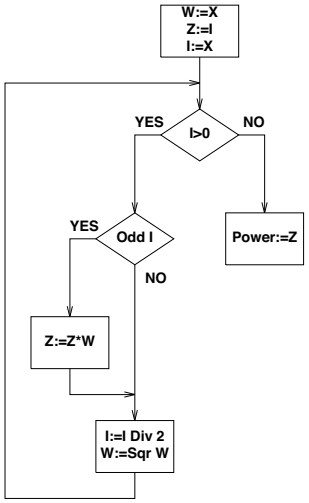
2. **use an input language that describes the algorithm and not the flowchart**

3. **need to input only the algorithm to get a standard flowchart**

Goals -4

4. **be able to add clues to flowchart definition in order to get a more customized flowchart, both globally applicable and locally applicable clues**

Icing: since there is a version, TPIC, of PIC that outputs TEX input, FLO can also be used with TEX.



Goals -5

So from the simple input:

```
.FL
[W:=X;
Z:=1;
I:=Y];
WHILE [I>0]; DO
BEGIN
   IF [Odd I];
      THEN [Z:=Z*W];
   [I:=I Div 2;W:=Sqr W];
END
[Power:=Z];
.FE
```

we get following default flowchart:

Goals -6

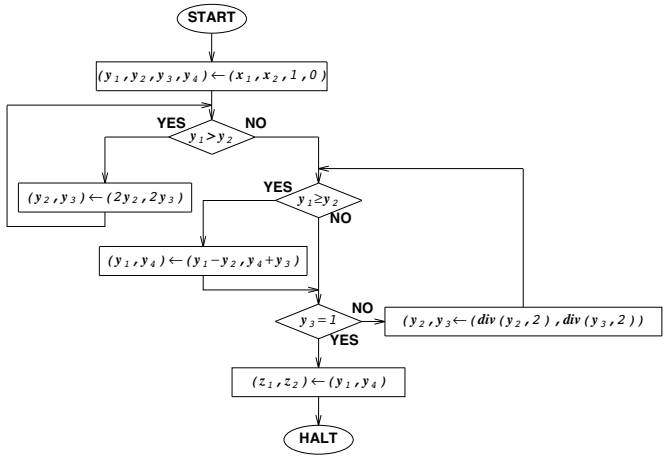
From the fancier input:

```
.FL
defshape ends shape is oval:
   {ellipse ht $1 wid $2} shapew is 0.6;
stmtshapeh is 0.25 ;
queryshapeh is 0.3 ;
spaceh is 0.25;
spacew is 0.2;
[START] with ends;
[(y1,y2,y3,y4)←(x1,x2,1,0)]
     shapew is 2.0;
WHILE [y1>y2];
   DO [(y2,y3)←(2y2,2y3)] shapew is 1.5;
```

```
LOOP
   IF [y1≥y2] ;
      THEN [(y1,y4)←(y1−y2,y4+y3)]
           shapew is 1.8;
   EXITIF [y3=1] config is RIGHT;
   [(y2,y3←(div(y2,2),div(y3,2))]
        shapew is 2.4;
   @up ;
END
[(z1,z2)←(y1,y4)] shapew is 1.5;
[HALT] with ends;
.FE
```

(For clarity, EQN input is shown already processed)

You can get an even fancier flowchart:



Concept -1

An includable flowchart is constrained to print on one page.

Its nodes are, therefore, a certain minimum readable size.

Therefore, the number of nodes in an includable flowchart is limited.

Thus, the complexity of an includable flowchart is limited.

Concept -2

Our algorithms do not have to handle all flowcharts

Exponential algorithms do not scare us.

Concept -3

We do not have to worry about interpage arcs.

We can focus on doing simple and small flowcharts well.

If user wants a multi-page flowchart, he or she must break it by hand into several one-page flowcharts.

Design Approach -1

1. We wrote an initial UM to describe all features that would be implemented.

Design Approach -2

We iterated on the initial UM until we were satisfied that the specified FLO could be used to draw all flowcharts that we had seen in recent books and papers on theory of computing and software engineering.

That is, the UM became the RS.

Design Approach -3

2. The first version of the UM used hand-coded PIC descriptions to draw each flowchart in the manual to look as if FLO had done it.

These hand-coded PIC descriptions were also our idea of the kind of output that FLO would create for the FLO input.

Design Approach -4

3. We did the following steps simultaneously and repeatedly:

- **We implemented features in FLO and commented out hand-coded examples in the manual source in order to let the FLO-generated PIC code show through.**

Design Approach -5

- **We modified the manual to reflect changes in FLO's implementation that were necessitated by discoveries made during failures to implement features as desired and as described in the manual.**

Design Approach -6

- **We modified the implementation to reflect changes in the manual that were indicated by discoveries made during attempted use of the features described in the manual to write a changed manual.**

Design Approach -7

The steps were repeated until we arrived at a manual and an implementation such that:

- **all implemented features were described in the manual,**
- **all features described in the manual were implemented, and**
- **all users are happy with the features and their output.**

Design Approach -8

As a bonus:

There was available at all times during the implementation a nice built-in test with full feature coverage.

The manual itself!

Design Approach -9

From the similarity in the structures of the papers and manuals about EQN, PIC, GRAP, DAG, and FLO, it *appears* that

the same approach was used by Kernighan, Bently, Cherry, and Trickey to design and implement EQN, PIC, GRAP, and DAG!

(When asked in a presentation of these slides, Kernighan said, "No!"

Design Approach -10

The roles in the development of FLO were:

I, the advisor, was the client, an occasional writer of computer science theory papers, representing all people who use flowcharts in formal papers.

I even occasionally asked a theoretician at the Technion, Nissim Francez, for his opinion on features.

Tony, the student, was the software engineer.

Design Approach -11

The initial manual underwent many, many iterations.

- **Any time I did not understand what it was saying, I complained.**
- **Any time I could not see the purpose of something, I complained.**
- **Many times, something it said suggested to me another option.**
- **Many times, something it said led to my asking how to do something related.**

Design Approach -12

Tony had to fix each of these problems, sometimes by changing or extending the language (almost never reducing the language! but...)

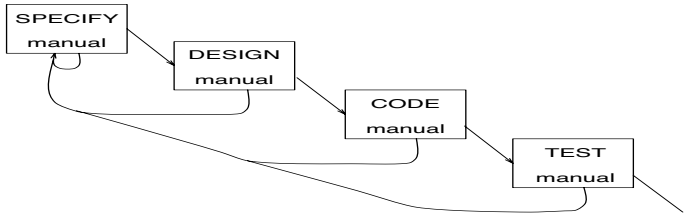
In one case, he threw out a whole collection of attributes, "short", "tall", etc. in favor of setting size of bubbles around nodes; bubbles turned out to be a simple way to specify completely the compactness of the layout.

Design Approach -13

The iteration continued until I could think of nothing wrong and nothing more!

Design Approach -14

FLO's development followed the waterfall lifecycle



Note the feedback into the requirements box (labeled "specify") and note that the manual gets worked on in all steps.

Wrap Up

Remember what this lecture is about!

It is my opinion that the UM serves as a useful elicitation, analysis, and validation tool, and it can even serve as a RS.

It was certainly the case for the design and implementation of FLO

WD-PIC Experiences

Let me describe 2 other experiences of using the UM for a program as its RS.

In each of these cases, the program has a graphical user interface (GUI) or a voice user interfact (VUI). Thus, the manual cannot be input directly to the program, and thus, the manual itself cannot be used as a test case for the program.

WD-PIC, a WYSIWYG Direct-Manipulation PIC

Faina Shpilberg
Daniel M. Berry, 1998

WD

"WD" stands for "WYSIWYG, Direct manipulation"

"WYSIWYG" stands for "What You See Is What You Get"

They are independent in principle, but usually come together in WIMP (Windows, Icons, Menus, Pointers) interfaces.

Batch vs WD Drawing Programs

Batch, e.g., PIC:

A can edit PIC specification with batch editor; insertion, global changes, working with groups, more convenient than with most WD drawing programs

D cannot see what you are doing

WD, e.g., xfig:

A can see what you are doing

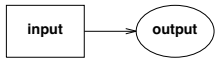
D painful to make insertions and global changes and work with groups

PIC

The input:

```
box "input"
arrow
ellipse "output"
```

yields

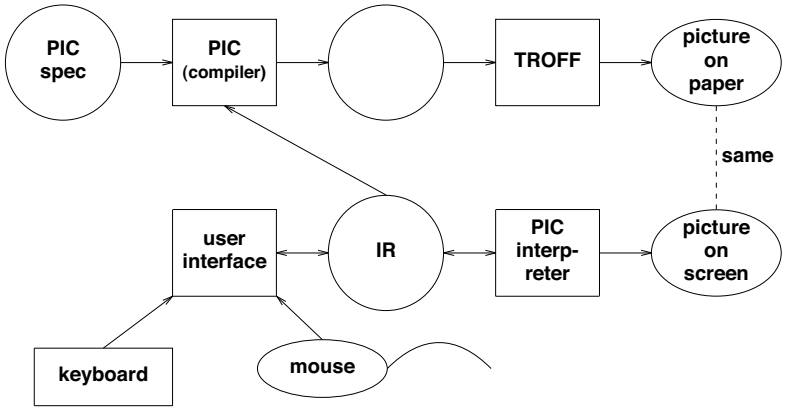


Goals of WD-PIC

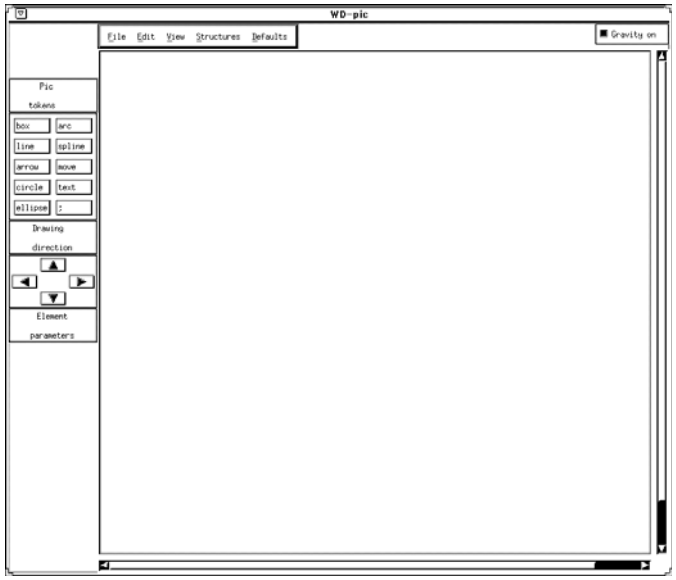
BOBW (Best of Both Worlds):

- **editable internal representation (IR), in the PIC language,**
- **can see the picture being drawn on the canvas,**
- **at any time the IR is a PIC specification of what is on the canvas,**
- **palette of PIC elements and menues for their attributes, and**
- **pull down menues for files, editing, views, and others.**

Batch PIC



WD-PIC Flow



WD-PIC vs. other WD Drawers

Because of equivalence of batch and WD picture for any IR,

in WD-PIC, rarely have to move mouse to canvas;

can stay in palette, clicking away.

In other WD drawers, must move mouse to canvas to position a clicked palette item.

Requirements -1

At any time, picture on canvas is same as printed on paper when the accumulated IR is submitted to PIC|TROFF.

At any time, clicking on *X* box is the same as entering "*X*" on the keyboard, for any PIC element *X*.

Requirements -2

E.g. all of the following are equivalent:

(stand-alone character is typed, labeled box is clicked)

b o x " i n p u t " ↵ a r r o w ↵

box " i n p u t " ↵ arrow

b o x " i n p u t " ↵ arrow

Requirements -3

Avoid dialogue boxes and confirmation buttons.

Cut, copy, paste, and other direct manipulation at the graphic level.

Requirements -4

IR of picture should be what human being would type, making use of defaults, and not showing full parameters with 8 digit floating point numbers as parameter values, e.g.:

```
box "input"
```

rather than

```
box wid .75237589 ht .58639282
at 3.8203785, 2.9851863 "input"
```

Requirements -5

(Note how these last two can contradict each other; moving a box to an arbitrary point requires pairs of 8 digit floating point numbers as coordinates.)

Grid of symbolic distances with origins in symbolically identifiable places, e.g., movewid × moveht grid centered at Box.ne.

Requirements -6

Can point to a grid point in order to, e.g., put things there by DM.

Can, at any time, edit the IR with the text editor associated with $EDITOR.

When save and exit from text editor, the picture is regenerated in the canvas.

These are the basic goals and requirements.

First Prototype

The first version was done by Shpilberg using prototyping method with Berry as the customer/user.

1. **Shpilberg implemented a version.**
2. **Berry tried it out.**
3. **Berry found something he did not like, and he complained.**
4. **Shpilberg understood the complaint, planned another change, and looped back to 1.**

Infinite Loop

Notice the infinite loop.

Berry never really liked it!

More on that below.

Prototyping -1

The method used for FLO cannot be used exactly since a manual cannot drive a WD program in the same way it can be the input to a batch formatter.

To make it easy to change, Shpilberg used a standard widget set for interaction objects.

Prototyping -2

These widgets forced use of dialogue windows and confirmation or cancellation and lots of hand movement to bring mouse into different windows and to click different buttons.

'Twas a royal pain, and it precludes the seamless arbitrary switching between batch and WD input.

Prototyping -3

Berry was never fully happy with the first version. He ended up using rapid sequences of button clicks, with no keyboard input or moving the mouse from the palette to rapidly build up a skeleton of the picture. Then he manually edits the saved IR to what he needs with a text editor.

Prototyping -4

It is still much much faster than inputting completely as text, which in turn is much much faster than specifying the entire picture in WD-PIC, especially if mistakes are made in entry.

Second Prototypes

Berry decided to have each team of a full-year SE studio at the Technion specify, design and implement its own enhancement of the legacy WD-PIC.

The first semester is devoted to requirements engineering to produce a requirements specification.

The second semester is devoted to designing, implementing, testing, and deploying the specified products.

Team's Job

Each team is

- **to program the new user interface in Java,**
- **to make use of knowledge of grammar to avoid use of dialogue windows and to allow in-line editing, and**
- **to re-use legacy PIC interpreter.**

Process

First, Berry

- **described and demonstrated the old software**
- **showed what he did not like, and**
- **gave them a long wish list.**

Then, they asked Berry questions in a two-hour long interview.

They continued to question Berry throughout term.

Results

Berry got at the end of the term 3 different WD-PICs and their UMs as specifications.

Less than Total Satisfaction

However, these were not exactly right for Berry as a customer, even though they were good student term projects that deserved A+s.

The reason was simply that an academic term project did not permit exploring requirements with the customer as thoroughly as does an open-ended project that continues until it is really done.

More WD-PIC UMs -1

In two consecutive iterations of a graduate seminar on requirements engineering at the University of Waterloo, Berry assigned as the term project writing a UM as a specification for WD-PIC.

Each team was given the source and object code and the UM for:

- **Shpilberg's WD-PIC**
- **the best of the 3 prototypes produced in the Technion SE studio**

More WD-PIC UMs -2

These projects yielded fine manuals and many interesting new ideas.

But, no one WD-PIC was completely satisfactory to Berry as a customer.

First Production Version

Lihua Ou took the assignment to produce a first production-quality version of WD-PIC as her master's thesis project.

Ou's Professional Background

Prior to coming to graduate school, Ou had built other systems in industrial jobs, mainly in commerce.

She had followed the traditional waterfall model, with its traditional heavy weight SRS.

She had made effective use of libraries to simplify development of applications.

Ou's Input

Ou was to look at all previous prototypes and UMs as specifications.

She was to filter these and scope them to first release of a production quality first version of WD-PIC running on Sun UNIX systems.

Ou's Assignment -1

Ou was to write a specification of WD-PIC in the form of a UM.

This UM was

1. **to describe all features as desired by the customer, and**
2. **to be accepted as complete by the customer,**

before she began any design or implementation.

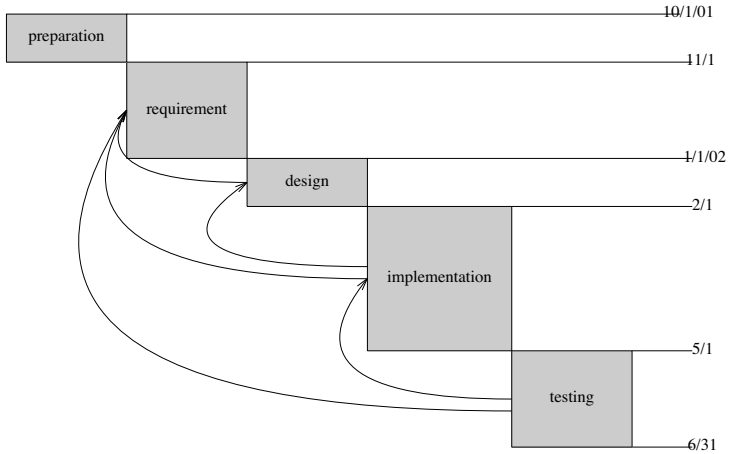
Ou's Assignment -2

Once implementation started, when new requirements are discovered, the manual should be modified to capture new requirements.

In the end, the manual describes the program as delivered.

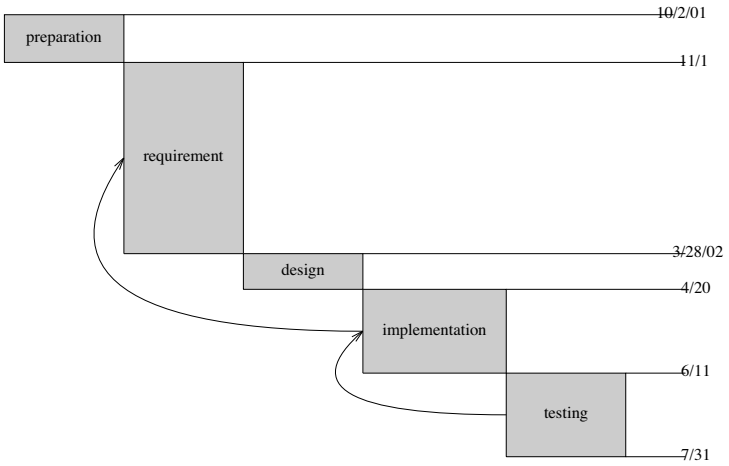
Project Plan

Duration in months	Step
1	Preparation
2	Requirements specification
4	Implementation
2	Testing
1	Buffer (probably more implementation and testing)
10	Total planned



Actual Schedule

Duration in months	Step
1	Preparation
4.9	Writing of user's manual = reqs spec, 11 versions
.7	Design including planning for maximum reuse of PIC code and JAVA library
1.7	Implementation including module testing and 3 manual revisions
1.7	Integration testing including 1 manual revision and implementation changes
10	Total actual



What Happened?

While detailed plan was not followed, total project time was as planned.

Also, Ou produced two implementations for the price of one, for:

- **(planned) Sun with UNIX and**
- **(unplanned) PC with Windows 2000**

Surprise

Ou was more surprised than Berry that she finished on time.

Berry had a lot of faith in the power of good RE to reduce implementation effort.

Adding to Ou's surprise was that the requirements phase took nearly 5 months instead of 2 months; the schedule had slipped 3 months out of 10, way beyond recovery.

Then and ...

Ou's long projected implementation and testing times and the 1 month buffer indicate that she expected implementation to be slowed by discovery of new requirements that necessitate major rewriting and restructuring.

Then and Now

This time, only minor rewriting and no restructuring.

Thus instead of 2 months specifying and 7 months implementing and testing,

she spent 5 months specifying and only 4 months implementing and testing.

Why?

By spending 3 additional months writing a specification that satisfied a particularly hard-nosed customer who insisted that the manual convince him that the product already existed,

Ou produced a specification that

- **had very few errors and**
- **that was very straightforwardly implemented.**

The Errors

Almost all errors found by testing were relatively minor, easy-to-fix implementation errors.

The two requirement errors were relatively low level and detailed.

They involved subfeatures in a way that required only very local changes to both the manual and the code.

What Helped?

All exceptional and variant cases had been worked out and described in the manual.

Thus, very little of the traditional

- **implementation-time fleshing out of exceptional and variant cases and**
- **implementation-time subconscious RE.**

Test Cases

The manual's scenarios, including exceptions and variants turned out to be a complete set of black box test cases.

Tests were so effective that, to our surprise, ...

scenarios not described in the manual, but which were logical extensions and combinations of those of the manual worked the first time!

The features composed orthogonally without a hitch!

Satisfied Customer

Berry found Ou's implementation to be production quality and is happily using it in his own work.

Industrial Case Study

For his master's degree research, Igor Fainchtein wrote a UM as a RS for ExpressPath.

ExpressPath

ExpressPath is a natural language speech recognition system developed by LGS, an IBM Company.

ExpressPath allows software to answer telephones and to satisfy user's requests in a voice-only interface.

Voice-only interface is more convenient to user than traditional Interactive Voice Response (IVR) system that uses the telephone's touch-tone keyboard for input.

ExpressPath is New -1

ExpressPath's voice user interface (VUI) is not well-known by the user community (like WD-PIC's paradigm).

Thus, a major part of the RE effort is the design and structure of the user interface.

No body of experience, guidelines, or standards upon which to draw.

ExpressPath is New -2

ExpressPath will set the standards.

Thus, difficulty understanding the UM raises concerns about the described system's usability

Fainchtein's Role

Fainchtein was a member of the team developing ExpressPath while studying for a master's degree and searching for an advisor and topic for his master's thesis.

After taking Berry's course, he approach Berry with an offer to apply the writing of a UM as the RS for ExpressPath.

Berry pounced on the offer as a way to get an industrial case study of his idea.

Project Needed RS -1

The project was in the early stages.

All of the company's previous projects in the domain of IVR had used RSs in the form of use cases agreed to by the customers and the developers.

Each use case was described in words and flowcharts, showing the use case's main, alternative, and exceptional scenarios.

Project Needed RS -2

Each scenario is a tree of menus and choices with

- **voice prompting by the application and**
- **DTMF input by the user.**

These use cases sufficed as RSs because the applications were in the well-understood IVR domain.

Project Needed RS -3

Over the years, they had developed and refined principles about IVR UIs.

For any new application, the customer understood any proposed solution without the need to consult each category of users directly.

Project Needed RS -4

ExpressPath with a VUI was in a totally new domain.

There was *no* user experience.

To get experience, the company decided to deploy its developing product experimentally on its own phone system.

Project Needed RS -5

Partly due to pressure to get the application out early and ...

partly due to the normal use of tacit assumptions, ...

the company was following its traditional requirements process, producing use cases as its RS.

Project Needed RS -6

Each use case was again a tree of menus and choices with

- **voice prompting by the application and**
- **voice input by the user.**

In other words, they were basically the same IVR application use cases with a change of input medium from DTMF to voice.

Project Needed RS -7

These use cases were accepted without reconsidering tacit assumptions and without consulting any category of users except developers, who were users of the in-house experimental deployment.

Project Needed RS -8

Occasionally these user-developers prototyped troublesome scenarios, but again without consulting any category of users but themselves.

However, the VUI was entirely new technology, less familiar to anyone.

Project Needed RS -9

Therefore, it was necessary to change the requirements process to one

1. **involving all categories of the users, and much more often,**
2. **rethinking the application from the ground up,**
3. **questioning tacit assumptions,**
4. **exploring entirely different alternatives, &**
5. **producing a RS that described the application thoroughly.**

Project Needed RS -10

Describing the application thoroughly would force consideration of all the details and questioning of tacit assumptions.

Fainchtein believed that writing a UM as a RS would be a good method to do what was needed and to get the desired RS.

Fainchtein's Offer

Fainchtein approached project leader with idea of producing a RS in the form of a UM.

Skeptical Manager

This idea appeared very risky to the skeptical manager.

However, manager understood that

- **some RS would be useful even though he did not perceive that**
  - **he had the resources for it or**
  - **it was valuable enough to commit his precious resources to it, and**
- **eventually, a UM would need to be written.**

The Deal

Fainchtein offered to write the UM on his own time.

Manager agreed that

- **Fainchtein would write a UM during time allotted to Fainchtein's studies, but**
- **Fainchtein could interact with project personnel and other stakeholders freely.**

The Payoff

Thus, the manager got a needed document

- **at a cost reduced to well below the document's perceived benefit,**
- **at a considerably reduced technical risk, and**
- **with a very high potential payoff in reduced costs, errors, etc.**

UM and Prototype

Because of prototyping, UM was not functioning as a poor man's prototype; it was serving as only the specification.

Some Prototyping Unnecessary

However, UM did make some prototyping unnecessary.

Once a prototype had introduced users to a particular paradigm, variations described by scenarios in the manual were clear and concrete enough to allow selection of the best without having to actually implement them in a prototype for user trials.

Existence of prototype made it easy to get screen snapshots for manual scenarios.

Time Required

Production of UM = RS took Fainchtein 5.5 months of part-time work, with full-time background thinking.

The company's experience with similar sized projects suggests that 5 months is needed for *each* of a SRS and a manual.

Thus, Fainchtein got one document serving as two documents for the price of one.

Time Savings

He saved time by writing only one document instead of two

It is clear in implementation so far, ...

that having clarified and documented requirements in advance of the implementation, the implementation is going more rapidly and error free than normally.

Customer Feedback -1

During writing of manual, Fainchtein got the usual number of complaints about specified requirements.

Each complaint resulted in a new iteration of the manual and resolution of the problem.

However, this resolution took place *during* requirements analysis time rather than during testing or after delivery.

Customer Feedback -2

The customer found the UM easier to read than the traditional SRS.

Consequently, the customer's complaints were more specific and more constructive and often contained a solution.

Team members other than Fainchtein found it easier than normal to elicit customer feedback using the UM.

Customer Feedback -3

Customer's validation was faster because he was able to see how certain functions were being used by a user.

The customer even said that having a manual instead of a traditional SRS contributed to a reduction in overall time spent to validate the requirements and allowed him to involve more domain experts and those who had never dealt with a traditional heavy SRS.

Manager's Testimonial -1

The skeptical project manager observed and confirmed that:

- **Prototyping and the UM allowed detecting in the *first* iteration more than 75% of all problems that they have found in the implementation.**

Manager's Testimonial -2

- **These problems were fixed immediately and the fixes were reflected in the next versions of the prototype and the UM.**
- **The UM was a source of test cases for the quality-assurance personnel.**

Customer's Testimonial

The customer was happy that the requirements process allowed the customer and developers to detect and readily address any human–computer interaction problem that arose during requirements specification.

Team's Testimonial -1

Analysis team found that having a RS in the form of a UM made it easier than normal to work with the customer to address human–computer interface (HCI) problems.

Team's Testimonial -2

Normally, a HCI problem would not even be detected until after the system were implemented.

Detecting HCI problems early is critical for any application with a new user interface paradigm.

Developer's Testimonial -1

The developers find the UM easy to work with as a RS.

The UM form of RS has made it easier new developers to get up to speed.

Developer's Testimonial -2

Having both the UM and the prototype has reduced the learning curve by at least 50% over having only a traditional SRS.

The developers have gotten into the habit of calling the UM "the specification".

Lessons Learned

Six Groups of Lessons Learned

- **advice on writing UMs in general and as RSs,**
- **special kinds of UMs,**
- **why UMs work as RSs,**
- **RE processes aided by writing a UM,**
- **other SE processes aided by writing a UM, and**
- **general lessons.**

Advice on writing UMs

You vs. User

There are two ways to describe the user in a UM:

- **"You" (second person) and imperative sentences with implied subject of "you"**
- **"the user" (third person)**

You vs. User, But Only One

In the last analysis, it does not matter which one you choose.

However, use *only* one; do *not* mix the two.

If you use both, the reader wonders if there are two kinds of users.

You vs. User Tradeoff

Second person is textually shorter than third person:

“You enter ‘exit’.” is shorter than
“The user enters ‘exit’.”,

and imperative:

“Enter ‘exit’.” is even shorter.

A Definite NO NO

If you use “the user”, remember that it is singular.

Therefore, the correct pronouns for it are:

“he”, “she”, and “he or she”

and NOT

“they”!

Grrrrrrrr!!!

Glossary of Terms

From the very beginning of the writing of a UM, you should build and *maintain* a glossary (dictionary, lexicon) of technical terms for the manual.

This glossary is not only for the benefit of the reader, but also for your benefit as the author to avoid two terrible scourges of writing that tries to be interesting,

- **US = unnecessary synonymy and**
- **CP = confusing polysemy**

Unnecessary Synonymy

US is using more than one word for the same concept, e.g., …

“the program”, “the software”, “the system”, “*X*” (where *X* is the name of the program), “the *X* software”, etc.

The reader is left wondering if there are even subtle differences between these different terms.

Necessary NonSynonymy

Occasionally you may need to distinguish between the software as an artifact supplied on a medium and an invocation of the software.

Necessary NonSynonymy, Cont'd

In that case, you make two entries into the Glossary:

"the *X* program = a copy of the *X* program on a CD" and

"*X* = an invocation of the *X* program running on your computer",

and you carefully maintain the distinction in the writing.

Confusing Polysemy

CP is using one word for more than one concept, e.g., …

Confusing Polysemy, Cont'd

One document that used "OS" to mean

- **operating system,**
- **an open source program, i.e., an artifact,**
- **open source software as a kind of software,**
- **open source development, i.e., a process, and**
- **the open source phenomenon, i.e., a metaprocess.**

Confusing Polysemy, Cont'd

It never talked about an open source operating system, which would be "OS OS", but even so, it was a *very confusing* document.

Note that the authors *knew* which meaning of "OS" they meant each time they used it, but the readers had to *guess*.

Plural of Acronyms

Be careful of acronyms in which an interior noun or an irregular noun gets pluralized:

"Request for proposals" = "RFP"

"Requests for proposals" = "RFPs"

"Brother-in-law" = "BIL"

"Brethren-in-law" = "BILs"

Plural of Acronyms, Cont'd

The pluralizing "s" comes at the end of the acronym even when it comes in the middle or not at all in the expansion of the acronym.

***Never* let an acronym be its own plural, even if its pronunciation is word that is its own plural, e.g.,**

"FISH" and "FISHs"

UMs Should Lie

The manual should be written well enough that it deceives the reader into believing that the software really exists.

In fact, it's getting the picky details worked out well enough to write the deceptive UM that forces ironing out those synergistic problems that plague many requirements, and even design, documents.

Faking it [Parnas & Clements, Simon & Garfunkel], again!!!

Present Tense

Important rule for any specification of a CBS:

Use present tense to talk about what the CBS *does*.

Consistent with faking it, that the CBS is already implemented.

Why Rule is Needed -1

Rule is needed so that:

When it is necessary to talk about something that happens in the user's future, after the user's present in which he or she says something to the CBS, future tense can be used to distinguish the future event from the user's input.

A typical specifier writes a specification for a not-yet-implemented product in the future tense, talking about what the CBS *will do*.

Why Rule is Needed -2

After the CBS is implemented in the future, the user will enter some input, and the CBS will do something in response.

The specifier loses ability to distinguish between the user's present and the user's future.

Everything happens in the specifier's future.

"Shall" vs. "Will"

There is a convention that is observed in many design and engineering disciplines for writing specifications:

When describing a requirement for the system to be built, say "The system shall …" …

and leave "will" to describe future events.

“Shall” vs. “Will”, Cont’d

In the vernacular, “shall” is often used as a future indicator with an additional compulsion component.

In specifications, “shall” is *reserved* for indicating requirements.

So be careful of how you use it!

Specifying “How” Information?

We are admonished to specify What a CBS does, and not How the CBS does it.

Sticking to What gives the implementers the greatest freedom.

Sometimes it is necessary to give some How details, usually under the rubric of “Architecture”.

It is possible to do so in a UM.

How Spec Example -1

E.g. Knuth exposed the line-breaking algorithm for TEX in *The TEXbook*, which serves both as a specification of TEX and as a UM, for 2 reasons:

1. **to ensure that *all* implementations of TEX produce the same formatted output, even down to the line breaks and spacing between words, and**

How Spec Example -2

2. **to give the user enough smarts about the line-breaking algorithm that he or she can exercise the commands to achieve the desired line breaking and interword spacing.**

That the specification of TEX is its UM served as a filter to make sure that a How detail showed up in the manual only when the detail is necessary for the user’s effective use of TEX.

Special kinds of UMs

Other Kinds of UMs

The above has talked about UMs as RSs for Information Systems.

There are other kinds of systems with no traditional UMs

- **Embedded, e.g, device controller or aircraft**
- **Platform, e.g. POSIX, X**

So it looks like advice fails

No! Just have to be creative at identifying users

Embedded System *E*

User of *E* is another system *S* with which *E* interacts.

Consider the programmer of *S* that uses *E*.

He or she needs to know what functions *E* offers under what conditions.

The manual for these functions of *E* is equivalent to a requirements specification for *E*.

Platform *P* -1

User of *P* is an application *A* running on top of *P*.

Consider the programmer of *A* that runs on top of *P*.

He or she needs to know what services *P* offers.

The manual for these services of *P* is equivalent to a requirements specification for *P*.

Platform *P* -2

Manual should be written as a system programmer's manual for *P*, e.g., a collection of UNIX manual pages for Sections 3 and 4, programs and data.

Guess what! That's exactly the format of the POSIX and X standards = specification. It is a specification in that anyone can implement these in whatever way he or she wants, so long as the specified external interface is met. These standards are written as the manual for any implementation.

Requirements for Platforms

A computing platform, e.g. operating system, has users that differ from the users of a single application.

A platform user is a sophisticated user who programs applications that run on the platform, for use by others.

Equivalence of RSs and UMs holds even for these platforms.

POSIX Example -1

E.g., the POSIX system is a standard generalization of the various UNIX platforms.

POSIX is specified by collection of UNIX-style man pages describing the kernel routines and data that are available to use to write applications running on any POSIX system.

POSIX Example -2

The man pages

- **describe what an implementation of a POSIX must make available; they give for each kernel routine the interface it must support.**
- **describe what an application running on a POSIX may assume; they give for each kernel routine the interface that can be assumed by invokers.**

UM for Platform

Even for an operating system, a well-written UM can serve as a RS.

Of course, this UM, aimed at the programmer of applications, may not appear well written to the user of an application.

UMs at Same Level as RS

The manual is an ideal requirements document in many cases because, if it is well-written,

- **it is written at the level of what the user sees,**
- **it describes the basic concepts, and**
- **it does not describe implementation details.**

That is, it is written at the right level of abstraction for a RS.

Why UMs work as RSs

Why UMs Validate Well

UMs seem easier to validate than traditional SRSs.

Why?

One group in the Technion SE studio insisted on writing a traditional SRS rather than the suggested UM.

They did a good job of it, following a standard template to the letter.

Comparing SRSs and Manuals

Thus, Berry had a chance to compare UMs for WD-PIC to a traditional SRS for WD-PIC.

Even though Berry is

- **thoroughly familiar with WD-PIC and**
- **thoroughly computer literate,**

he had a hard time understanding some specifications of some features in the SRS.

SRS vs. UM -1

With SRS, Berry could not see that what was specified was not quite what he wanted.

With UM, Berry had no such problems; He was able to instantly spot specifications that did not capture his desires and to describe what was wrong and how to fix it or at least what the misunderstanding was.

SRS vs. UM -2

The clarity of the two specifications were like night and day.

Berry empathized with customers who report that they understand and accept specifications that they were too embarrassed to admit that they had not understood at all.

Key Difference -1

The normal SRS describes only what the system does.

The UM describes conversations between user and system to achieve user's goals.

The UM was more alive.

Berry could see himself as the user in the manual.

Key Difference -2

He could spot instantly described user behavior that did not correspond to what he would do.

The normal SRS does not describe user's inputs and reactions.

It describes only system's behavior in a vacuum from the user.

Key Difference -3

So, Berry had no idea what user behavior was implied.

Thus, if a behavior bore any resemblance to his preconceived idea of system behavior, he believed that the SRS described what he wanted.

Designers and Implementers

Can be argued that a UM favors the user over the designer and implementer.

If a UM is use-case centered, it's hard to identify functions that must be implemented.

A Good UM

However, a good UM has also a feature-centered part listing all the individual features and describing all of the options of each.

This part is organized much as is a traditional SRS.

The designer and implementer can find the functions already identified.

UMs & RSs are Difficult

Writing the UM is hard, but so is writing a RS!

So you are no worse off!

All the Gory Details -1

No requirements specification method that does not force working out the details is going to work.

It is only in working out the details that all the show-stopping exceptions and interactions are going to be discovered.

All the Gory Details -2

These details can be worked out in any of several media:

- **the software itself,**
- **a complete formal specification,**
- **a complete, traditional SRS, or**
- **a complete, scenario-based UM.**

The advantage of UM is that changing the manual consistently is much cheaper than changing either the software itself or a complete formal specification.

All the Gory Details -3

Also, unlike a complete, traditional SRS, a UM is both needed and perceived as needed after the software is delivered.

Thus, the motivation to keep it up to date is higher than that to keep a traditional SRS up to date.

All the Gory Details -4

The advantage of the software itself or a complete formal specification is that it is hard to handwave over the details, to cheat to leave the impression of completeness when details are missing.

If details have been left out, the software will not work or the formal specification cannot be verified to satisfy requirements.

All the Gory Details -5

While it is fairly easy to leave details out of a UM, since the UM is intended to be delivered with the software to help naive users, the incentive is to get those details in.

Thus, it is an issue of finding a right medium for expressing detailed requirements that is both cheap to change, but hard to hand wave one's way to a false impression of completeness.

Writing a UM Helps RE

UMs Help Validation

Thus, giving a draft UM to the client and to the client's users is a good way to elicit the

"But, I wanted x instead"s

and the

"But, I wanted x also"s

before the software is implemented.

i.e., to validate the requirements!

UMs Help Elicitation

The UM actually serves a dual purpose.

- **aiding the elicitation of correct and complete requirements**
- **being the requirements document**

UMs and Prototyping

Sometimes in order to write a UM, you have got to prototype or at least mock up the screens so that you can get the nice screen pictures that you want to put in the manual.

This prototype also helps in getting the client to validate the requirements.

Scenario Capture

In the SE studio for WD-PIC, the first structured exercise was to develop a full set of scenarios to capture everything they thought Berry would want to do.

Berry noticed a systematic approach that discovered about 80% of the scenarios.

Scenario Generation by Grammar

Most scenarios consist of a user creating an individual PIC element adjusted with no or a variety of attributes.

Scenario Generation -2

Thus, it seems reasonable to follow the context-free grammar of the intermediate rep, i.e., the PIC language itself and

- **to regard the sublanguage for each PIC item as a high level scenario that can call subscenarios,**
- **to regard each way to input a token, e.g. by mouse click or keyboard entry, as a subscenario, and**

Scenario Generation -3

- **to allow a digression, i.e., a subscenario, before and after each terminal and nonterminal**
  - **to do some file command,**
  - **to edit the IR,**
  - **to change the view,**
  - **to change session attributes,**
  - **to specify grid, gravity, or both,**
  - **to set defaults,**
  - **etc.**

Systematics

The systematics of this approach helps insure completeness.

It does not guarantee completeness, unfortunately.

Finding Ambiguities -1

Our experience in the case studies, particularly the WD-PIC projects, is that the process of writing and validating a UM used as a specification helps expose ambiguities, even the subconscious ones.

Finding Ambiguities -2

Student groups clarified a number of misunderstanding with Berry by showing him a UI and a number of use cases.

Berry's reactions to the UI and the use cases made the ambiguities and his intent clear and helped to disambiguate the ambiguities.

Writing a UM Helps SE

Scenarios as Test Cases

Notice that we used the FLO manual as its own test case.

More generally, scenarios or use cases, can be used to generate test cases.

Equivalence of Ss and TCs

A little thought shows a fundamental equivalence between scenarios and test cases

Both must cover all possible inputs and uses of the software under consideration, i.e., the software under design or test.

Obtaining this full coverage is hard.

The literature on testing abounds with proof of this difficulty.

```
.FL
IF [blah];
   THEN BEGIN
      IF [bluh] ;
         THEN [y];
         ELSE [z];
      END
   ELSE BEGIN
      IF [bleah];
         THEN [b];
         ELSE [c];
      END
[d];
.FE
```

Example of Difficulty

Indeed, after FLO had been used about a year with no problems, an input was given that FLO drew incorrectly, collapsing two boxes into one, with their texts overwriting each other:

Overlooking an Obvious TC or S

It turned out that we had *never* tested that particular input or any of the infinitely others that showed the problem.

There were no such examples in the manual!

In retrospect, it is hard to imagine not having nested conditionals as a scenario or as a test case if the goal is complete coverage.

However, neither of us ever thought of it.

Overlooking -2

Perhaps we did not view this as a special case, because we had tried other forms of nesting.

Moreover, it's so common that no one else thinks of it as very special.

It showed up one day when we were using FLO for a production job (Isn't that how *all* bugs show up?!?)

Conclusion:

Completeness of scenarios and completeness of test cases are tough to achieve, very tough!

By the way, FLO was fixed in a second release! Sigh.

Test Case Generation -1

While the manual of a WD-program cannot be a test case itself, the scenarios provide scripts for test cases.

These can be run by humans following the scripts,

Test Case Generation -2

OR

each can be translated into a process that generates the sequence of events that would be generated by a human following the scripts,

the latter providing automatic re-run of test cases whenever it is necessary.

Integral Help -1

An old idea, revisited:

"Self-Describing Systems Using Integral Help" by Bob Fenchel and Jerry Estrin, IEEE Trans. Systs, Man, & Cybernetics, March/April, 1982.

Integral help (IH) for interactive, textual-input / textual-and-pictorial-output system design shell

Integral Help -2

Fenchel took the context-free grammar for the shell language and put a "?" before each non-terminal and terminal in each production.

X → A x B y

is converted to

X → ?A?x?B?y

The shell interpreter considers "?" matched either by itself or by the empty string.

Integral Help -3

If the user inputs a "?", then at the very least the interpreter responds with the portion of the original grammar that follows the matched "?".

E.g., if in the production above, the input "?" matched the first "?", the shell would respond with at least

expecting A x B y

Integral Help -4

The system designer could also provide for any particular "?" a more detailed response, written by a human being to deal with the probable *real* question at that point.

From Help to Scenarios -1

We can go further.

Before each terminal and nonterminal in the grammar, we can have invocations of all possible subscenarios.

Each subscenario describes a different way to get the next input token, e.g., by typing, by clicking a button.

From Help to Scenarios -2

Each subscenario describes a different way to adjust the background and state of the object being worked on, e.g., place or remove grid and turning gravity on or off, etc.

This gives a systematic way to discover scenarios for an interactive system building objects defined by a predefined representation, e.g., PIC, RTF, etc.

General Lessons

Requirements Notations -1

As you are writing a RS, you may use whatever organization and notation that helps you achieve these objectives.

Requirements Notations -2

When picking the organization and notation, remember the intended audience.

This means not using a notation that will turn your audience off.

On the other hand, this does not mean to shy away from a notation from fear that the audience will not understand it.

Requirements Notations -3

It is my experience that any notation

- **that suits the situation in which it is used**
- **that is consistently applied**

can be learned by anyone of reasonable intelligence that is in a position to understand what is being conveyed, e.g., a client.

Requirements Notations -4

After all, the software is being developed for the client's application domain, and he or she clearly knows the vocabulary of that application — the professional jargon.

Requirements Notations -5

For example, a state machine is a very natural way to describe reactive systems, but is not a good way to describe calculation of the standard error of the difference between two vectors, and is not a good way to describe the formatting of a paragraph.

For the client of a reactive system, a state transition diagram would make perfect sense, while for the statistician or the word processing specialist, a state transition diagram would appear as mumbo-jumbo.

Requirements Notations -6

A control and data flow diagram is a very natural way to describe an industrial process in which there may be several activities happening concurrently.

For the client of a process control system, a control and data flow diagram would make perfect sense, while for the statistician or the word processing specialist, a control and data flow diagram would appear as gobbledegook.

Requirements Notations -7

My experience has been that a good notation just happens during the elicitation and analysis phase.

It happens because at the time it was introduced, it seemed like the logical thing to use.

Requirements Notations -8

When it is introduced in this way, the notation is understood instantly or with minimum explanation by all involved in the elicitation and analysis.

The notation writes itself!

Quotations Worth Quoting

Never underestimate the lack of sophistication of the unsophisticated user!

Never underestimate the lack of sophistication of the *sophisticated* user!

The donut from X was *so* bad that the best tasting part was the hole!

Conclusions

Writing a good RS is hard.

It is hard to motivate people to write one.

UM can be ideal RS

A UM is an ideal RS in many cases because, if it is well-written,

- **it is written at the level of what the user sees,**
- **it describes the basic concepts, and**
- **it does not describe implementation details.**

That is, it is written at the right level of abstraction for a RS.

Case Studies Show

Certainly, the case studies have demonstrated that for appropriate CBS, the production of a UM and the UM itself can serve the purposes of and provide the benefits of producing a RS and the RS itself.

However, the issue remains: whether producing a UM helps mitigate the inhibitions against producing a RS.

Academic Case Studies

In academic case studies, the professor was a dictator who could force producing a RS even if the students hated it! ☺

Industrial Case Study -1

In the industrial case study, there was not going to be a suitable RS and there was not even going to be a totally suitable RE process.

The project leader was not motivated at first to even try producing a UM as the RS.

Only when he got an offer, that could not be refused, of a UM written on an employee's own time, with minimal resource demands on employees working for him, that he agreed to an attempt to produce a UM as the RS.

Industrial Case Study -2

Adding to the deal's sweetness was the realization that a UM would eventually have to be written anyway.

In accepting the offer, perhaps as a favor to Igor's education that he was already supporting, he remained skeptical as to the UM's effectiveness as a RS.

Industrial Case Study -3

However, once it became apparent to the leader and to the rest of the team that ...

writing the user's manual was working as an RE process and that they would get both an acceptable RS and a UM for the price of one, the entire project team, including the leader, bought into the UM as "the spec".

Perhaps More Can Try It

With enough demonstrations of effectiveness and cost benefits, perhaps more projects can be persuaded to adopt this approach as an effective compromise between the extremes of having no RE process at all and having a full-fledged heavy-weight RE process producing both a SRS and related documents and a UM.

Wary of YAMBW

While industry is wary of jumping on yet another methodological bandwagon (YAMBW), it does adopt techniques that are repeated demonstrated in practice to be effective, such as inspection.

Perhaps with enough studies like these showing clear success stories, the UM-as-RS approach will be adopted.

Successful Case Studies -1

These case studies of writing RSs in the form of UMs are just case studies.

Certainly, each case study was a success in that

- **the writing of the UM helped focus requirements elicitation and analysis,**
- **the writing of the UM forced consideration of user needs in the product and in its requirements,**

Successful Case Studies -2

- **the UM provides a covering set of test cases,**
- **in each case in which the product has been delivered, the UM describes the product completely,**
- **in each case in which the product has been delivered, the customer is satisfied with the end product and the match between the UM and the product,**

Successful Case Studies -3

- **in one case, the UM helped the programmer meet the overall schedule even though writing the UM caused a schedule slip in the RE phase, and**
- **in the case in which there was not going to be a fully documented RS, the developers have ended up calling the UM "the requirements".**

Threats in Academic Cases

However, these successes in the non-industrial cases could have been the result of other factors including

- **the abilities of the programmer and the client,**
- **the client's previous experience with the application,**
- **the UI centeredness of the application, and**
- **the medium size of the application.**

Threats in Industrial Case

These were not factors in the industrial case. In that case, the success could have been the result of

- **the Hawthorne effect, since writing a UM as a RS was a totally new and experimental activity for the people involved, and**
- **the fact that it was at first not considered a real part of the project, because it was being done as an academic project on Igor's own time**

Cannot Generalize

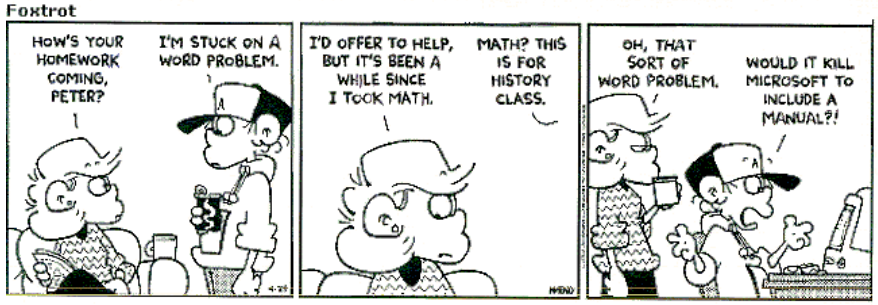
Thus, we cannot necessarily conclude that UMs will always make good RSs and that writing the UM as the RS will always help a project to be successful.

Hard to Generalize in SE

There is no completely satisfactory to validate any SE method. There's a trade off between statistical significance and generalizability to real-life, industrial strength projects.

More Industrial Studies Needed

So, more case studies are needed to build up an experience base to led ever-increasing credibility to the claim that UMs make good RSs for the appropriate kinds of CBSs.


Foxtrot
HOW'S YOUR HOMEWORK COMING, PETER?
I'M STUCK ON A WORD PROBLEM.
I'D OFFER TO HELP, BUT IT'S BEEN A WHILE SINCE I TOOK MATH.
MATH? THIS IS FOR HISTORY CLASS.
OH, THAT SORT OF WORD PROBLEM.
WOULD IT KILL MICROSOFT TO INCLUDE A MANUAL?!

But No One Makes UMs Any More

♩♪♩ Oh where, oh where have all the manuals gone?

I am reminded of a Foxtrot comic strip that has Peter, who is trying to do a History homework, screaming "Would it kill Microsoft to include a manual?"

True, But ...

It's true, many companies do not produce UMs any more.

However, they *do* make help systems, and these provide a good covering set of scenarios.

The help system can be used, as we suggest, as the RS.

Help Systems vs. UMs

In a sense, the scenarios of a good help system are better than most UMs for our purposes because each scenario focuses on one way that users use the software described by the help system to do their work.

User's Manual as a Requirements Specification

Daniel M. Berry, 1991, 1998, 2002, and 2003 with help from K. Daudjee, J. Dong, I. Fainchtein, M.A. Nelson, T. Nelson, & L. Ou

© 2003 Daniel M. Berry

Advice on writing UMs

This is page 170 in the older document

You vs. User

There are two ways to describe the user in a UM:

- **"You" (second person) and imperative sentences with implied subject of "you"**
- **"the user" (third person)**

You vs. User, But Only One

In the last analysis, it does not matter which one you choose.

However, use *only* one; do *not* mix the two.

If you use both, the reader wonders if there are two kinds of users.

You vs. User Tradeoff

Second person is textually shorter than third person:

“You enter ‘exit’.” is shorter than
“The user enters ‘exit’.”,

and imperative:

“Enter ‘exit’.” is even shorter.

A Definite NO NO

If you use “the user”, remember that it is singular.

Therefore, the correct pronouns for it are:

“he”, “she”, and “he or she”

and NOT

“they”!

Grrrrrrrr!!!

Glossary of Terms

From the very beginning of the writing of a UM, you should build and *maintain* a glossary (dictionary, lexicon) of technical terms for the manual.

This glossary is not only for the benefit of the reader, but also for your benefit as the author to avoid two terrible scourges of writing that tries to be interesting,

- **US = unnecessary synonymy and**
- **CP = confusing polysemy**

Unnecessary Synonymy

US is using more than one word for the same concept, e.g., …

“the program”, “the software”, “the system”, “*X*” (where *X* is the name of the program), “the *X* software”, etc.

The reader is left wondering if there are even subtle differences between these different terms.

Necessary NonSynonymy

Occasionally you may need to distinguish between the software as an artifact supplied on a medium and an invocation of the software.

Necessary NonSynonymy, Cont'd

In that case, you make two entries into the Glossary:

"the *X* program = a copy of the *X* program on a CD" and

"*X* = an invocation of the *X* program running on your computer",

and you carefully maintain the distinction in the writing.

Confusing Polysemy

CP is using one word for more than one concept, e.g., …

Confusing Polysemy, Cont'd

One document that used "OS" to mean

- **operating system,**
- **an open source program, i.e., an artifact,**
- **open source software as a kind of software,**
- **open source development, i.e., a process, and**
- **the open source phenomenon, i.e., a metaprocess.**

Confusing Polysemy, Cont'd

It never talked about an open source operating system, which would be "OS OS", but even so, it was a *very confusing* document.

Note that the authors *knew* which meaning of "OS" they meant each time they used it, but the readers had to *guess*.

Plural of Acronyms

Be careful of acronyms in which an interior noun or an irregular noun gets pluralized:

"Request for proposals" = "RFP"

"Requests for proposals" = "RFPs"

"Brother-in-law" = "BIL"

"Brethren-in-law" = "BILs"

Plural of Acronyms, Cont'd

The pluralizing "s" comes at the end of the acronym even when it comes in the middle or not at all in the expansion of the acronym.

***Never* let an acronym be its own plural, even if its pronunciation is word that is its own plural, e.g.,**

"FISH" and "FISHs"

"Shall" vs. "Will"

There is a convention that is observed in many design and engineering disciplines for writing specifications:

When describing a requirement for the system to be built, say "The system shall …" …

and leave "will" to describe future events.

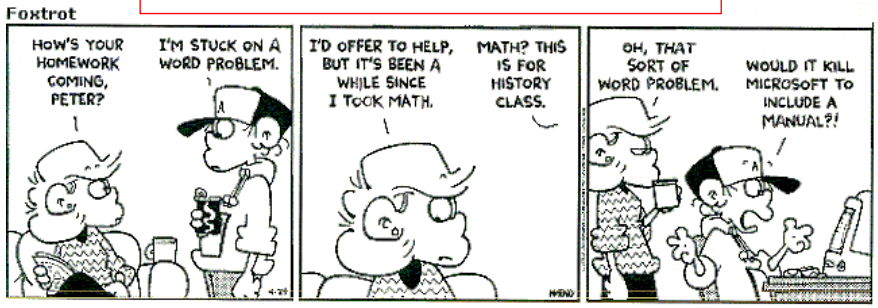
“Shall” vs. “Will”, Cont’d

In the vernacular, “shall” is often used as a future indicator with an additional compulsion component.

In specifications, “shall” is *reserved* for indicating requirements.

So be careful of how you use it!

This comes right after the second last page in the older document.


Foxtrot
HOW'S YOUR HOMEWORK COMING, PETER?
I'M STUCK ON A WORD PROBLEM.
I'D OFFER TO HELP, BUT IT'S BEEN A WHILE SINCE I TOOK MATH.
MATH? THIS IS FOR HISTORY CLASS.
OH, THAT SORT OF WORD PROBLEM.
WOULD IT KILL MICROSOFT TO INCLUDE A MANUAL?!

UMs Should Lie

This is page 171 in the old document

The manual should be written well enough that it deceives the reader into believing that the software really exists.

In fact, it’s getting the picky details worked out well enough to write the deceptive UM that forces ironing out those synergistic problems that plague many requirements, and even design, documents.

Faking it [Parnas & Clements, Simon & Garfunkel], again!!!